

Institut Préparatoire aux Études d'Ingénieurs de Tunis



Leila Ben Othman

Deuxième année de classes préparatoires

SM - SP

Année universitaire: 2018 - 2019

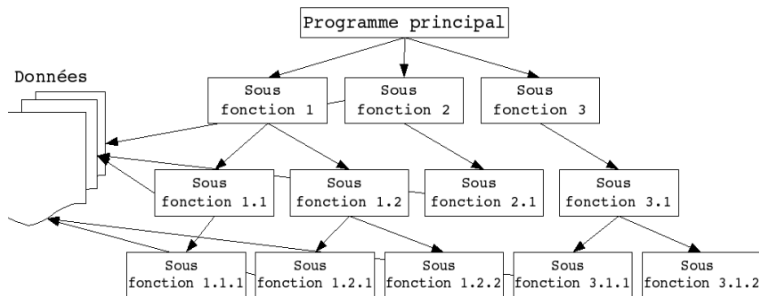
La Programmation Orientée Objet en PYTHON

Plan

- ① Introduction
- ② Les concepts de base de l'approche objet
- ③ PYTHON: langage de programmation orienté Objet

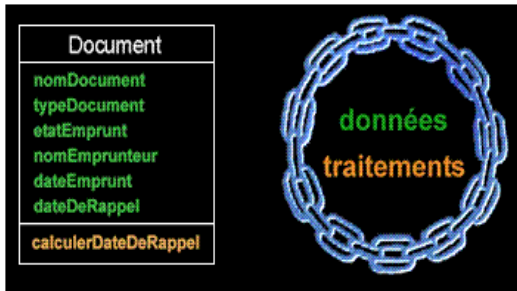
L'approche fonctionnelle/procédurale

- Dans une approche fonctionnelle, les programmes sont décomposés en une hiérarchie de fonctions et procédures.
- L'approche fonctionnelle se base sur une séparation entre les données et les traitements.



L'approche objet - Principe

- Principe : Rassembler les données et leurs traitements associés au sein d'une même entité.



⇒ Nouvelle approche de modélisation et de programmation

L'approche objet

Qu'est-ce qu'on peut modéliser?

- une personne, un livre
- un cercle, un carré, un rectangle
- un point, un vecteur, un polynôme
- un atome, etc

⇒ et plus encore!

L'approche objet

Nous allons nous intéresser à ces notions relatives à l'approche objet:

- Encapsulation
- Classe
- Attributs, méthodes
- Objet, instance
- Instanciation
- Héritage
- Polymorphisme, redéfinition, Surcharge

Les concepts de base de l'approche objet - Encapsulation

- L'**encapsulation** est un mécanisme consistant à rassembler les données (attributs) et les traitement (méthodes) au sein d'une même entité.
- L'encapsulation permet de définir des **niveaux de visibilité** des éléments de cette entité. Ces niveaux de visibilité définissent les **droits d'accès** aux données (publique, protégé, privé) ⇒ **Hors programme**

Les concepts de base de l'approche objet - Classe

- Dans la description d'un problème réel, il est impossible de décrire tous les objets du domaine. Les classes sont un moyen pour créer des objets qui présentent une **structure commune** (objets ayant les mêmes caractéristiques).
- Une classe est un **modèle** qui décrit plusieurs objets.
- Les caractéristiques de la classe s'appellent des **attributs**.
- Les actions qu'on peut appliquer sur un objet sont appelées des **méthodes**.

NB: On définit les attributs et les méthodes au niveau de la classe.

Les concepts de base de l'approche objet - Classe

Exemple :

- On peut définir une classe **Personne** avec comme attributs **nom** et **prenom**
- `modifier_nom`, `modifier_prenom`, `retourner_nom` et `retourner_prenom` peuvent être définies comme des méthodes de la classe **Personne**

NB: Evidemment, on aurait pu retenir d'autres caractéristiques (attributs) et d'autres méthodes pour la classe **Personne**. Cependant, ceci nous suffit pour notre exemple.

Les concepts de base de l'approche objet - Objet - Instanciation

- La relation qui existe entre une classe et un objet est une relation d'**instanciation** : un objet est une **instance** (occurrence) de classe.
- Des instances (objets) possibles de la classe **Personne**:
 - Personne1(nom='Ben Salah',prenom='Salah')
 - Personne2(nom='Mistiri',prenom='Farah')

⇒ Quand on instancie (crée) des objets, on définit des valeurs aux attributs.

PYTHON: langage de programmation orienté Objet

- PYTHON est un langage orienté Objet
⇒ Tout est objet : données, fonctions, modules, etc
- Un objet PYTHON possède:
 - un type (nature de l'objet): la fonction **type** permet de retourner le type d'un objet (**classe**).
 - une valeur (des données matérialisées par des valeurs d'**attributs**)
 - des opérations applicables (**méthodes**)

Exemple - PYTHON

- Tous les types qu'on a manipulés (*int*, *float*, *str*, *complex*, *list*, *tuple*, etc) sont en réalité définis en tant que classes.

Instanciation d'un objet de type `complex`

```
> x=1+5j ou bien x=complex(1,5)
> type(x)
<class 'complex'>
```

Attributs de la classe `complex`

```
> x.real
1.0
> x.imag
5.0
```

Méthode de la classe `complex`

```
> x.conjugate()
(1-5j)
```

Exemple - PYTHON

- Dans l'exemple précédent, notez bien l'utilisation d'un attribut par la syntaxe `x.nom_attribut` et d'une méthode par la syntaxe `x.nom_methode()`. Évidemment, une méthode peut avoir un ou plusieurs paramètres.
- Attention: Ne pas confondez pas méthode et fonction.
- Pour avoir le module d'un nombre complexe, on utilise la fonction `abs` (qui est une fonction donc sa syntaxe d'appel diffère de la syntaxe d'appel d'une méthode).

Module d'un nombre
complexe

```
> abs(x)  
5.0990195135927845
```

Définition de classe - syntaxe

```
class Personne:  
    def __init__(self):  
        #constructeur de la classe Personne  
        #avec parametres par défaut  
        self.nom='Ben_Salah'  
        self.prenom='Salah'
```

Définition de classe - syntaxe

- **class** est un mot clé permettant de définir la classe.
- `Personne` est le nom de la classe (par convention, commence par une lettre majuscule).
- N'oubliez pas les « : » pour l'indentation.
- `def __init__(self)` est une méthode spéciale appelée **constructeur**, automatiquement appelée lors de l'instanciation d'un objet.
- Au niveau du constructeur, nous avons défini deux attributs `nom` et `prenom` dont les valeurs seront attribuées au moment de l'instanciation (ici des valeurs par défaut).
- Le paramètre **self** (convention) représente l'instance elle-même, il doit apparaître comme premier paramètre dans toutes les méthodes. Mais, on ne le précise pas lors de l'appel.

Instanciation d'objets

```
#Instanciation – Creation de deux objets  
#de la classe Personne  
P1=Personne()  
P2=Personne()
```

- `P1=Personne()`: permet de créer une première instance de la classe `Personne` $\Rightarrow$ appel au constructeur `__init__` de la classe `Personne`.

NB: On ne précise pas le paramètre `self` du constructeur lors de l'instanciation.

```
print ( P1 . nom , P1 . prenom )  
print ( P2 . nom , P2 . prenom )
```

- Les deux instances ont les mêmes valeurs Ben Salah Salah (les valeurs par défaut affectées aux attributs au niveau du constructeur).
- Notez bien le rôle de «.» lors de l'accès aux attributs de l'objet.

NB: Les attributs sont accessibles en lecture/écriture. Mais il n'est pas conseillé de faire ça en orienté objet. On accédera aux attributs via des méthodes.

```
class Personne:
    def __init__(self ,a ,b):
        #version 2 du constructeur de la classe
        #Personne avec parametres
        self .nom=a
        self .prenom=b

#Instanciatiion –Creation de deux objets
#de la classe Personne
P1=Personne( 'Ben□Salem ' , 'Marwa ' )
P2=Personne( 'Mistri ' , 'Farah ' )
```

Cette fois, lors de l'instanciation il faut passer les paramètres (valeurs des attributs nom et prenom) de pour chaque personne créée.

```
class Personne:
    def __init__(self, a, b):
        self.nom=a
        self.prenom=b
    def afficher(self):
        print('Nom:␣', self.nom)
        print('Prenom:␣', self.prenom)
    def modifier_nom(self, a):
        self.nom=a
    def modifier_prenom(self, b):
        self.prenom=b
    def retourner_nom(self):
        return self.nom
    def retourner_prenom(self):
        return self.prenom
```

Testez vos méthodes!

```
P1=Personne( 'Ben_Salem ', 'Marwa ' )  
P1.modifier_prenom( 'Salima ' )  
P1.afficher()  
print( 'Le_nom_est_',P1.retourner_nom())
```

Les concepts de base de l'approche objet - Héritage

- L'**héritage** est un mécanisme de transmission des caractéristiques d'une classe (attributs et méthodes) vers une sous-classe.
- L'héritage implique :
 - une relation de **généralisation** : regroupement au sein d'une classe les caractéristiques communes à un ensemble de classes.
 - une relation de **spécialisation** : ajout de caractéristiques propres à une sous-classe.
- La généralisation et la spécialisation génèrent une **hiérarchie de classes**.

Héritage - Exemple

- Dans la suite de notre exemple, on va définir deux sous-classes: classe Etudiant et classe Enseignant qui héritent tous les attributs et méthodes de la classe Personne.
- On ajoutera les attributs (num et matière) comme attributs spécifiques ainsi que des méthodes spécifiques à chaque sous-classe.

Héritage en PYTHON

```
class Etudiant(Personne):  
    def __init__(self, a, b, number):  
        Personne.__init__(self, a, b)  
        self.num=number  
  
    def afficher(self):  
        Personne.afficher(self)  
        print('Num: ' + str(self.num))  
  
    def modifier_num(self, number):  
        self.num=number  
  
    def retourner_num(self):  
        return(self.num)
```

Héritage en PYTHON

```
class Enseignant(Personne):  
    def __init__(self, a, b, mat):  
        Personne.__init__(self, a, b)  
        self.matiere=mat  
  
    def afficher(self):  
        Personne.afficher(self)  
        print('Matiere:␣',self.matiere)  
  
    def modifier_mat(self, mat):  
        self.matiere=mat  
  
    def retourner_mat(self):  
        return(self.matiere)
```

Les concepts de base de l'approche objet - Polymorphisme

- Le **polymorphisme** représente la faculté d'une méthode à pouvoir s'appliquer à des objets de classes différentes.
- Deux types de polymorphismes:
 - Polymorphisme d'héritage
 - La surcharge

Les concepts de base de l'approche objet - Redéfinition

- Dans notre exemple, on a réécrit la méthode **afficher()** pour adapter l'affichage en fonction de l'objet: s'il s'agit d'un enseignant, on affiche en plus sa matière. S'il s'agit d'un étudiant, on affiche en plus son numéro. $\Rightarrow$ On parle de **redéfinition** de méthode.
- Le **polymorphisme** d'héritage désigne la possibilité de redéfinir une méthode dans des classes héritant d'une classe de base.
- `O.afficher()` pourra, grâce au polymorphisme d'héritage, effectuer l'affichage adéquat en fonction de la classe de l'objet référencé au moment de l'appel (selon le type de l'objet `O`).

Les concepts de base de l'approche objet - Surcharge

- La surcharge (type de polymorphisme) permet d'avoir des méthodes portant le même nom agissant de manières différentes dans des classes sans aucun rapport entre elles (ceci n'est pas le cas de PYTHON, car toutes les classes héritent de la classe object).
- La surcharge permet ainsi de définir des opérateurs dont l'utilisation sera différente selon le type des objets.
- Par exemple, il est possible de surcharger l'opérateur "+" et de réaliser des actions différentes selon qu'il s'agisse d'une opération mathématique entre deux entiers (addition) ou d'une opération de concaténation entre deux chaînes de caractères.

Les concepts de base de l'approche objet - Surcharge

En PYTHON, il est possible de surcharger plusieurs méthodes, citons par exemple:

- `__repr__`: surcharge de *print(Objet)*
- `__add__`: surcharge de `de " + "`
- `__getitem__`: surcharge de *Objet[i]*
- `__call__`: surcharge de *Objet(i)*

NB: voir la surcharge de ces méthodes en TD - exercice Polynôme

La méthode `__repr__`

- La méthode `__repr__(self)` est une méthode spéciale de représentation qui permet l'affichage d'un objet chaque fois que la fonction **print** est invoquée
⇒ permet la surcharge de la fonction **print**: la méthode `afficher()` devient inutile.
- La méthode `__repr__(self)` doit retourner une chaîne de caractères qui sera affichée avec **print**.

La méthode `__repr__`

Personne

```
def __repr__( self ):  
    return ( 'nom : ' + self . nom +  
            ' _ prenom : ' + self . prenom )
```

Etudiant

```
def __repr__( self ):  
    return ( Personne . __repr__( self ) +  
            ' _ num : ' + str ( self . num ) )
```

La méthode `__repr__`

Enseignant

```
def __repr__( self ):  
    return ( Personne.__repr__( self ) +  
            ' _matiere: _ ' + self.matiere )
```